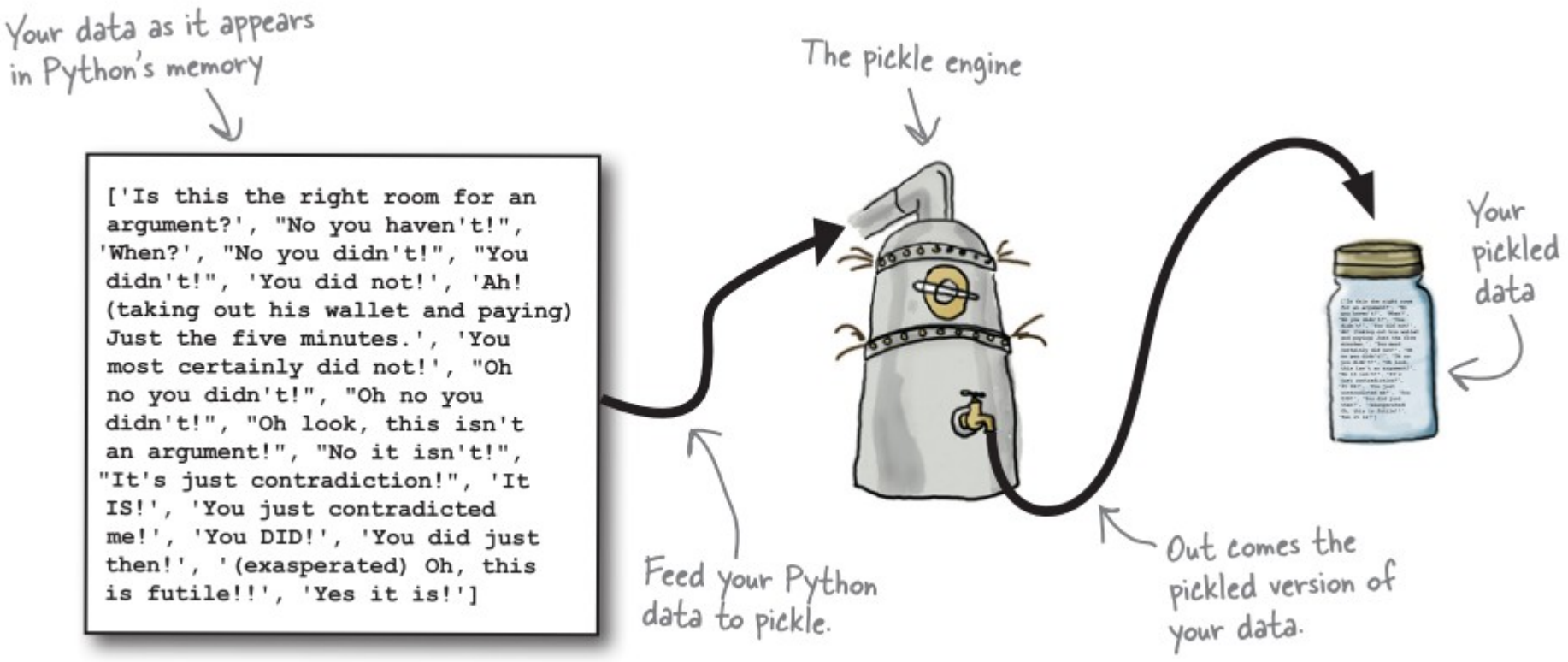
Сериализация

десериализация


PYTHON SERIALIZATION
WITH PICKLE

Pickle - алгоритм сериализации и десериализации объектов Python.



Сериализация/десериализация

Сериализация (англ. **serialization**) — процесс перевода какой-либо структуры данных в любой другой, более удобный для хранения формат. Обратной к операции сериализации является операция десериализации (англ. **deserialization**) — восстановление начального состояния структуры данных из битовой последовательности.

Самой основной структурой данных в языке программирования Python является объект. Сериализация и десериализация объектов используется в том случае, если нам надо передавать информацию между запусками одной программы или между несколькими программами.

Способы сохранить/восстановить объект.

1. Pickle
2. JSON
3. YAML

pickle

Модуль pickle реализует мощный алгоритм сериализации и десериализации объектов Python. "Pickling" - процесс преобразования объекта Python в поток байтов, а "unpickling" - обратная операция, в результате которой поток байтов преобразуется обратно в Python-объект. Так как поток байтов легко можно записать в файл, модуль pickle широко применяется для сохранения и загрузки сложных объектов в Python.

Часто сериализация используется для сохранения пользовательских данных между разными сессиями работы приложения, обычно игры.

Что можно консервировать ?

- None, True или False
- Числа , вещественные и комплексные числа
- Строки , байт строки.
- Списки, наборы , картежи и словари
- Функции определенные def в голобальной области
- Классы определенные в глобальной области
- Экземпляры классов у которых можно вызвать __dict__

Интерфейс модуля `dir(pickle)`

```
['ADDITEMS', 'APPEND', 'APPENDS', 'BINBYTES', 'BINBYTES8', 'BINFLOAT',
'BINGET', 'BININT', 'BININT1', 'BININT2', 'BINPERSID', 'BINPUT',
'BINSTRING', 'BINUNICODE', 'BINUNICODE8', 'BUILD', 'BYTEARRAY8',
'DEFAULT_PROTOCOL', 'DICT', 'DUP', 'EMPTY_DICT', 'EMPTY_LIST',
'EMPTY_SET', 'EMPTY_TUPLE', 'EXT1', 'EXT2', 'EXT4', 'FALSE', 'FLOAT',
'FRAME', 'FROZENSET', 'FunctionType', 'GET', 'GLOBAL', 'HIGHEST_PROTOCOL',
'INST', 'INT', 'LIST', 'LONG', 'LONG1', 'LONG4', 'LONG_BINGET',
'LONG_BINPUT', 'MARK', 'MEMOIZE', 'NEWFALSE', 'NEWOBJ', 'NEWOBJ_EX',
'NEWTRUE', 'NEXT_BUFFER', 'NONE', 'OBJ', 'PERSID', 'POP', 'POP_MARK',
'PROTO', 'PUT', 'PickleBuffer', 'PickleError', 'Pickler', 'PicklingError',
'PyStringMap', 'READONLY_BUFFER', 'REDUCE', 'SETITEM', 'SETITEMS',
'SHORT_BINBYTES', 'SHORT_BINSTRING', 'SHORT_BINUNICODE', 'STACK_GLOBAL',
'STOP', 'STRING', 'TRUE', 'TUPLE', 'TUPLE1', 'TUPLE2', 'TUPLE3',
'UNICODE', 'Unpickler', 'UnpicklingError', '_Framer',
'_HAVE_PICKLE_BUFFER', '_Pickler', '_Stop', '_Unframer', '_Unpickler',
'__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_compat_pickle',
'_dump', '_dumps', '_extension_cache', '_extension_registry',
'_getattribute', '_inverted_registry', '_load', '_loads', '_test',
'_tuplesize2code', 'bytes_types', 'codecs', 'compatible_formats',
'decode_long', 'dispatch_table', 'dump', 'dumps', 'encode_long',
'format_version', 'io', 'islice', 'load', 'loads', 'maxsize', 'pack',
'partial', 're', 'sys', 'unpack', 'whichmodule']
```

Методы модуля

Модуль pickle предоставляет следующие методы, чтобы сделать процесс pickling (пиклинг) более удобным:

Синтаксис:

pickle.dump(obj, file, protocol=None, *, fix_imports=True, buffer_callback=None)

Записывает сериализованный объект в файл. Дополнительный аргумент protocol указывает используемый протокол. По умолчанию равен 3 и именно он рекомендован для использования в Python 3 (несмотря на то, что в Python 3.4 добавили протокол версии 4 с некоторыми оптимизациями). В любом случае, записывать и загружать надо с одним и тем же протоколом.

```
import pickle
obj = {"one": 123, "two": [1, 2, 3]}
fd = open("data.pkl", "wb")
pickle.dump(obj, fd, pickle.HIGHEST_PROTOCOL)
```

pickle.dumps

Возвращает обработанное представление объекта obj в виде байтового объекта,без записи его в файл.

Синтаксис:

```
pickle.dumps(obj, protocol=None, *, fix_imports=True,
             buffer_callback=None)
```

Параметры:

```
obj – объект Python, подлежащий сериализации,
file – файловый объект
protocol=None – протокол сериализации
fix_imports=True   – сопоставление данных Python2 и Python3
buffer_callback=None – сериализация буфера в файл как часть потока
pickle.
```

Пример:

```
import pickle
obj = {"one": 123, "two": [1, 2, 3]}
output = pickle.dumps(obj, 2)
```

pickle.load

Восстанавливает сериализованный объект из файла

Синтаксис:

```
pickle.load(file, *, fix_imports=True,
encoding="ASCII", errors="strict")
```

Версия протокола определяется автоматически

Считайте байтовое представление объекта из открытого файла и возвращает сериализованный объект.

Пример:

```
import pickle

with open("data.pkl", "rb") as f:

    obj = pickle.load(f)
```

pickle.loads

Восстанавливает сериализованный объект в обычное представление из байтового.

Синтаксис:

```
pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")
```

```
import pickle

obj = {"one": 123, "two": [1, 2, 3]}

output  = pickle.dumps(obj)

new_obj = pickle.loads(output)
```

JSON Serialization

Пользоваться pickle лучше только в рамках python-приложения. При обмене данными между разными языками программирования обычно используют модуль json. Также pickle никак не решает вопрос безопасности данных. Поэтому не следует десериализовать данные из неизвестных источников.

Для работы с форматом JSON в Python используется модуль json

```
import json
```

Итерфейс модуля `dir(json)`

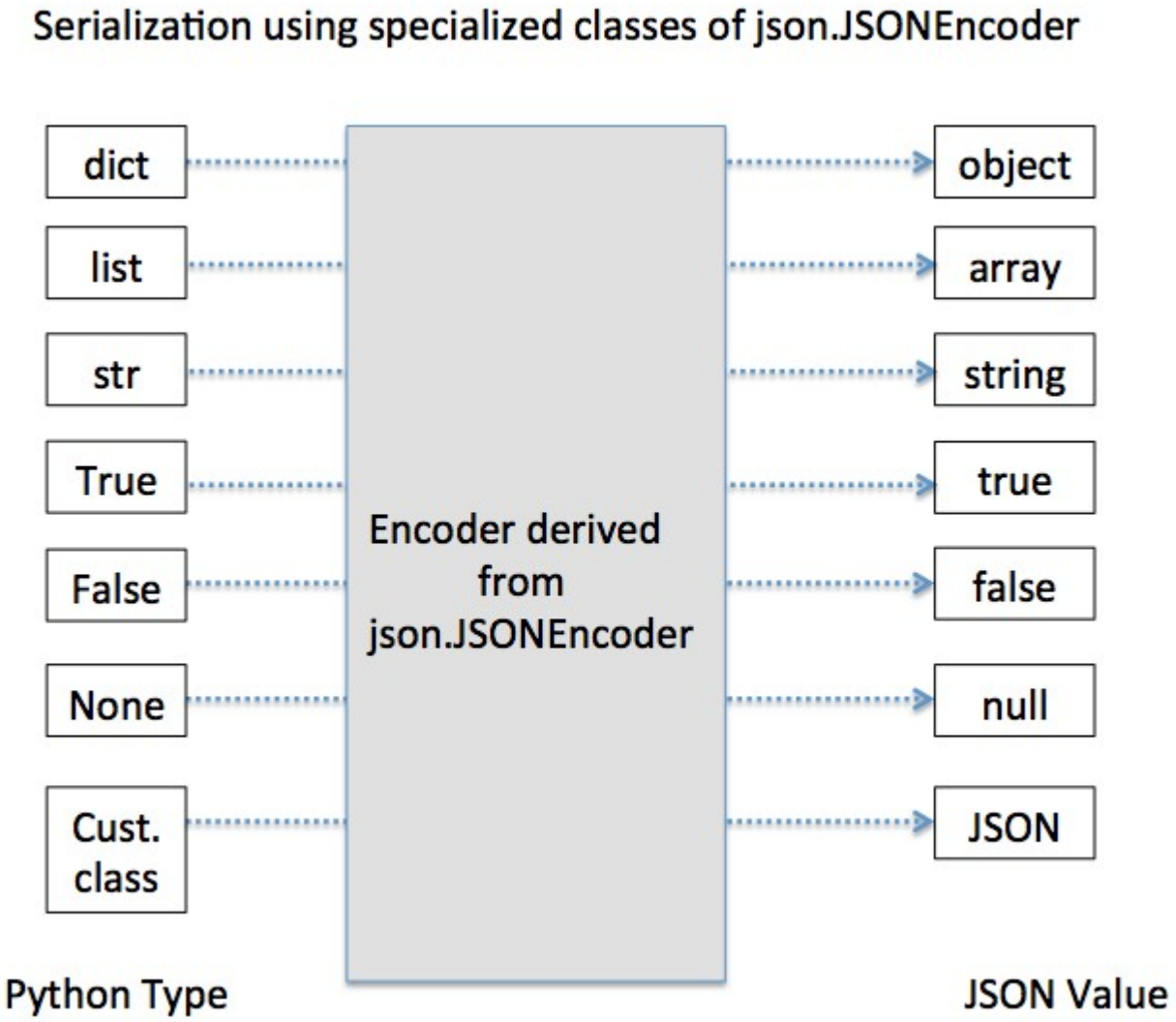
```
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder',
'__all__', '__author__', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__path__', '__spec__', '__version__',
'_default_decoder', '_default_encoder', 'codecs',
'decoder', 'detect_encoding', 'dump', 'dumps',
'encoder', 'load', 'loads', 'scanner']
```

Сериализация и десириализация

Сериализация JSON

Модуль json предоставляет удобный метод dump() для записи данных в файл. Существует также метод dumps() для записи данных в обычную строку. Типы данных Python кодируются в формат JSON в соответствии с интуитивно понятными правилами преобразования, представленными в виде таблице ниже.

Кодировщики и декодировщики



Сериализация и десериализация в строку

json.dumps(*object*) – сериализует obj в строку JSON-формата

json.loads(*output*) – десериализует строку содержащую документ JSON в объект Python.

```
import json
obj = {"one": 123, "two": [1, 2, 3]}

output  = json.dumps(obj)
new_obj = json.loads(output)
```

Во что превратится tupl – ?

Сериализация в файл

json.dump(obj, ff, indent="") - сериализует объект в файл.

```
import json
obj = {"one": 123, "two": [1, 2, 3]}

with open("data.json", "wt") as f:
    json.dump(obj, f, indent=4)
```

indent – количество отступов при записи

Десериализация из файла

json.load(*file_descriptor***)** – десериализует содержимое файла

```
import json
with open("data.json", "rt") as f:
    obj = json.load(f)
```

Десериализация из файла

```
import json

with open("data.json", "rt") as f:
    obj = json.load(f)
```

YAML Serialization

YAML — это язык для сериализации данных, который отличается простым синтаксисом и позволяет хранить сложноорганизованные данные в компактном и читаемом формате.

Для работы с форматом YAML в Python используется модуль yaml

Установка

```
$ pip install pyyaml
```

Подключение

```
>>>import yaml
```

Итерфейс модуля `dir(yaml)`

```
[ 'StreamEndToken', 'StreamStartEvent', 'StreamStartToken',
'TagToken', 'Token', 'UnsafeLoader', 'ValueToken',
'YAMLError', 'YAMLObject', 'YAMLObjectMetaclass',
'__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__',
'__spec__', '__version__', '__with_libyaml__', '_yaml',
'add_constructor', 'add_implicit_resolver',
'add_multi_constructor', 'add_multi_representer',
'add_path_resolver', 'add_representer', 'compose',
'compose_all', 'composer', 'constructor', 'cyaml', 'dump',
'dump_all', 'dumper', 'emit', 'emitter', 'error', 'events',
'full_load', 'full_load_all', 'io', 'load', 'load_all',
'loader', 'nodes', 'parse', 'parser', 'reader',
'representer', 'resolver', 'safe_dump', 'safe_dump_all',
'safe_load', 'safe_load_all', 'scan', 'scanner',
'serialize', 'serialize_all', 'serializer', 'tokens',
'unsafe_load', 'unsafe_load_all', 'warnings']
```

Сериализация и десериализация

yaml.dump(obj) – сериализация объекта в строку

yaml.load(new_obj) – десериализация объекта в строку

```
import yaml
from yaml.loader import SafeLoader

obj = {"one": 123, "two": [1, 2, 3]}
output = yaml.dump(obj)
new_obj = yaml.load(output, Loader=SafeLoader)
```

Сериализация в файл

```
import yaml


obj = {"one": 123, "two": [1, 2, 3]}
with open("data.yml", "wt") as f:
    yaml.dump(obj, f)
```

Десериализазация из файла

```
import yaml


with open("data.yml", "wt") as f:
    obj = yaml.load(f)
```

Заключение

Сериализация и десериализация объектов Python является важным аспектом распределенных систем. Вам часто приходится взаимодействовать с другими системами, реализованными на других языках, а иногда просто нужно сохранить состояние своей программы в постоянном хранилище.

Python поставляется с несколькими схемами сериализации в своей стандартной библиотеке, а многие другие доступны в качестве сторонних модулей.